Productionizing predictive models

GoDataFest – Open Source

Niels Zeilemaker

GO DATA DRIVEN

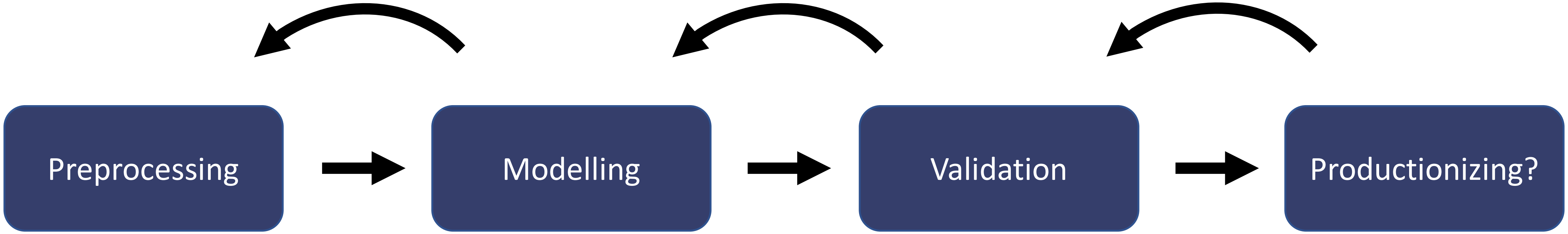
This hackaton

- ~15 minutes introduction
- ~1-1,5 hours hackathon / demo
- ~15 minutes wrap-up
- Plan to finish around 11:45

The machine learning process

GO DATA DRIVEN

The machine learning process



Preprocessing

Modelling

Validation

Productionizing?

Exploring and cleaning the data

~60% effort

Picking the right model

~10% effort

Validating if the model solves the problem

~30% effort

Getting the model where it can be used

100-???% effort

Productionizing ML models

What is productionizing?

Productionizing

- Goal – convert model into a (standard) format that can be run in production

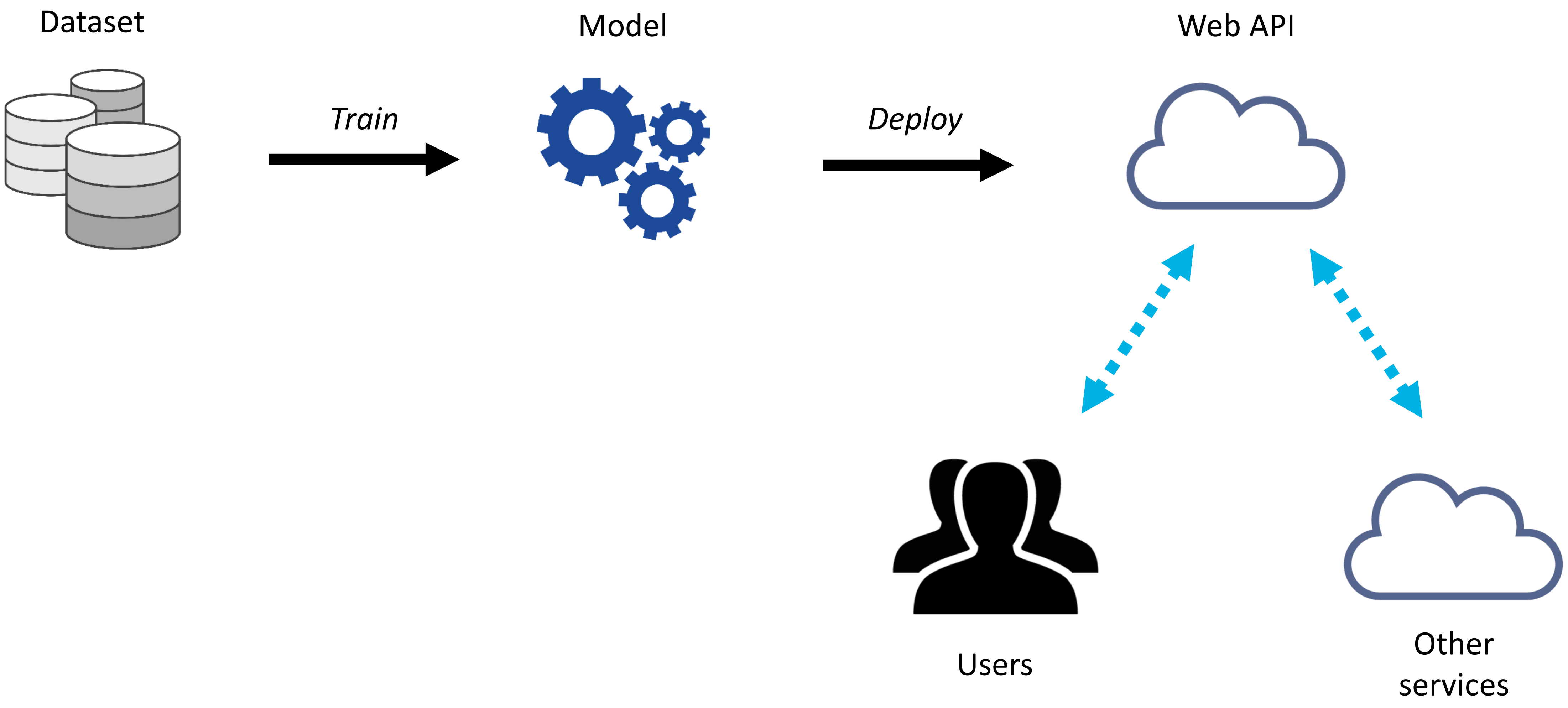
- How – depends on the production environment
  - Re-write code into a (production-quality) library
  - Wrap in an API for interfacing with other components

Productionizing

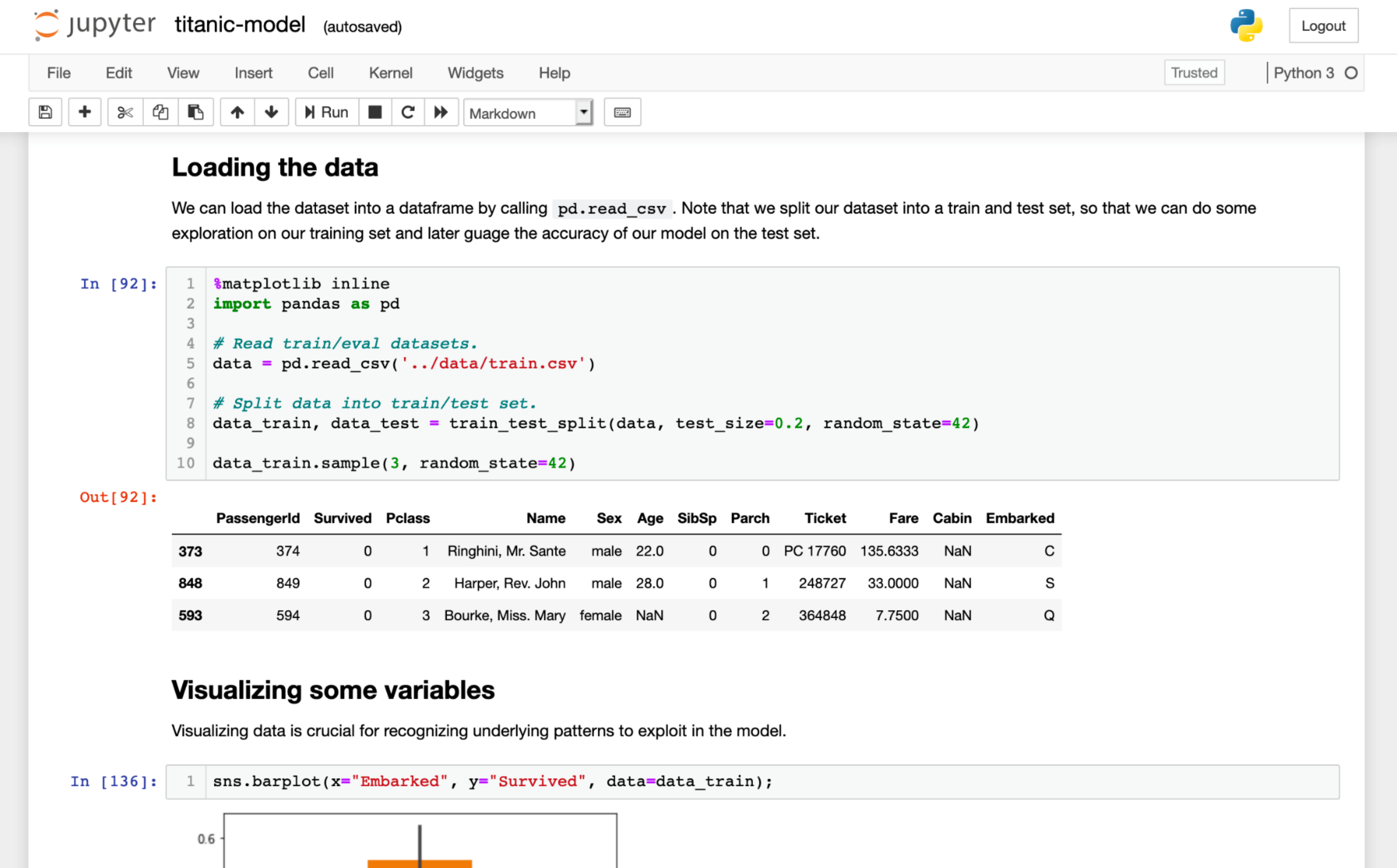
- Follow best practices
  - Version control
  - Code quality checks, unit testing
  - Logging / monitoring

- Consider deployment patterns
  - How will the model learn and predict?
  - What will we expose to the outside world?

Example: web-based API



However, many models look like this



Loading the data

We can load the dataset into a dataframe by calling `pd.read_csv`. Note that we split our dataset into a train and test set, so that we can do some exploration on our training set and later guage the accuracy of our model on the test set.

In [92]:

```
%matplotlib inline
import pandas as pd

# Read train/eval datasets.
data = pd.read_csv('../data/train.csv')

# Split data into train/test set.
data_train, data_test = train_test_split(data, test_size=0.2, random_state=42)

data_train.sample(3, random_state=42)
```

Out[92]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
373	374	0	1	Ringhini, Mr. Sante	male	22.0	0	0	PC 17760	135.6333	NaN	C
848	849	0	2	Harper, Rev. John	male	28.0	0	1	248727	33.0000	NaN	S
593	594	0	3	Bourke, Miss. Mary	female	NaN	0	2	364848	7.7500	NaN	Q

Visualizing some variables

Visualizing data is crucial for recognizing underlying patterns to exploit in the model.

In [136]:

```
sns.barplot(x="Embarked", y="Survived", data=data_train);
```

How do we move this into production?

- Start building a Python package
  - Isolate main components, move these into modules
  - Identify building blocks -> make reusable functions/classes

- Improve code quality
  - Implement quality checks (pylint) and tests (pytest)
  - Document code (docstrings) and package (readme, etc.)

- Wrap model in an API (Python, Flask)

Hackathon

- Background
  - Client interested in upselling cruise ship tickets
  - Noticed that in the titanic disaster, people in higher ticket classes had a higher chance of survival
  - Would like to present this information during the booking process to sell more 1st class tickets

Hackathon

- Scenario
  - Data scientist has created a model predicted survival probabilities based on the titanic dataset
  - We have been asked to move his/her notebook into production
- Goal – build a documented + tested Python package that exposes the model as a web API

Hackathon

- Getting started
  - Go github.com/godatadriven/code-breakfast-productionizing, read the README
  - Setup a clean Python environment and

    install the packages in notebook/requirements.txt
  - Try running the notebook and see if you understand its contents
- Afterwards - continue with the Step 2 (see readme)

Python packaging

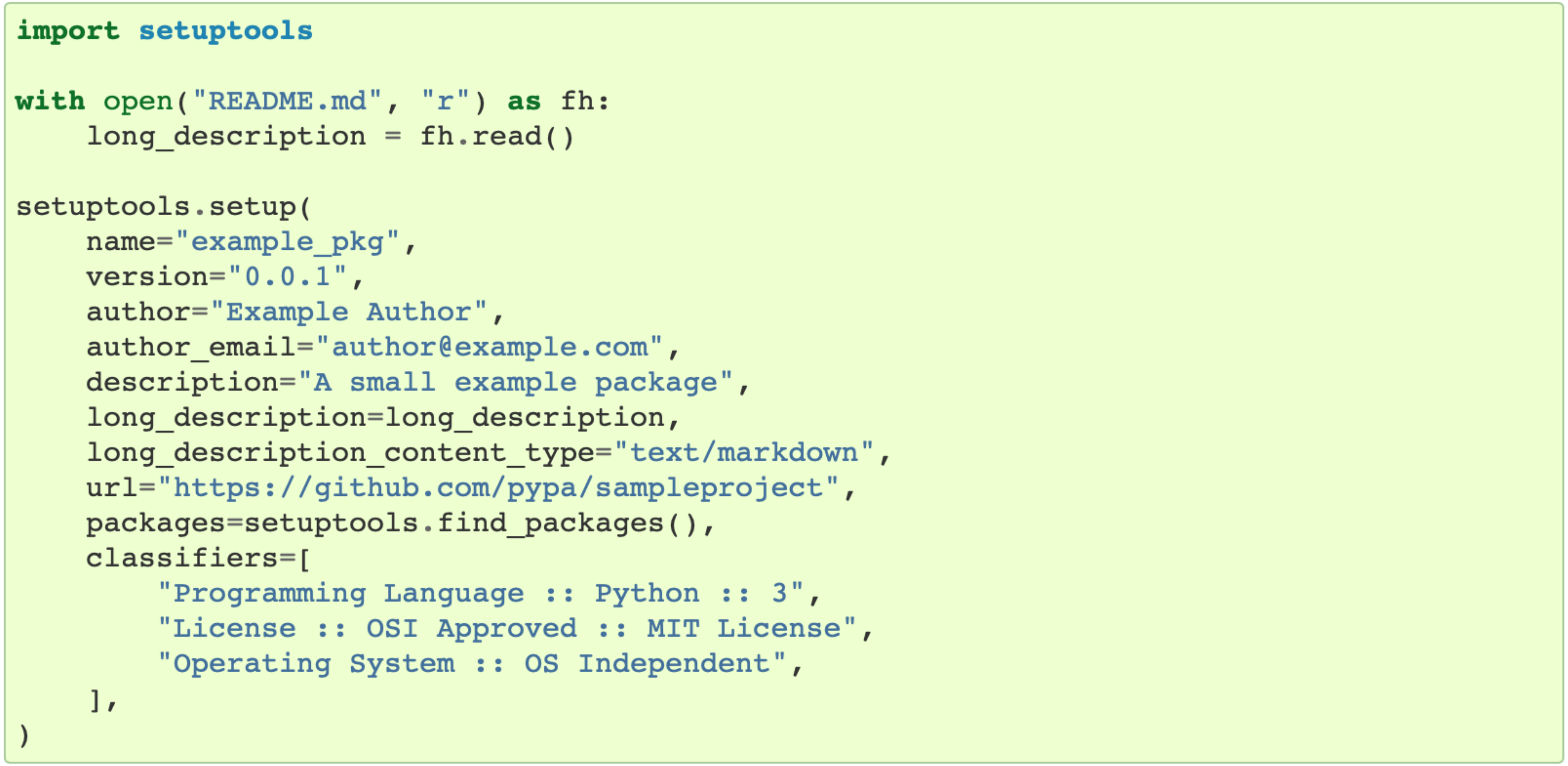
GO DATA DRIVEN

Python packaging

- Goal – package Python code into a redistributable package that can easily be installed in other environments

- Terminology
  - package – a directory with an __init__.py
  - module – a something.py file
  - subpackage – a package within a package

Setup.py



```python
import setuptools

with open("README.md", "r") as fh:
    long_description = fh.read()

setuptools.setup(
    name="example_pkg",
    version="0.0.1",
    author="Example Author",
    author_email="author@example.com",
    description="A small example package",
    long_description=long_description,
    long_description_content_type="text/markdown",
    url="https://github.com/pypa/sampleproject",
    packages=setuptools.find_packages(),
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
)
```

A simple example

```
├── LICENSE
├── README.md
├── my_package
|   ├── __init__.py
|   ├── my_module.py
|   ├── second_module.py
|   └── subpackage
|       ├── __init__.py
|       └── another_module.py
└── setup.py
```

```
import my_package
from my_package import my_module
from my_package import second_module
from my_package import subpackage
from my_package.sub_package import another_module
```

Additional functionality

- Besides code, Python packages also typically contain:
  - Documentation (Sphinx)
  - Unit/integration tests (Unittest library or pytest)
  - Additional readme/configuration files
- We will go into these later

Installing your package

- Your package can easily be installed using pip:
  - `pip install .`

- During development, you can use an editable install:
  - `pip install --editable .`

- This way, edits are directly reflected in your environment.
  (Note: notebooks require the autoreload extension.)

Additional resources

- Python tutorial: modules
- Python tutorial: packaging Python projects
- A tour on Python packaging
- Python's new package landscape
- Pypackage cookiecutter template

Code quality

Code quality

Code is a means to communicate not only with machines but also with other developers. High quality code is good communication.

High quality code is:

• Correct	runs correctly
• Human readable	is easy to understand
• Consistent	same formatting and naming
• Modular	small units of logic
• Reusable	code can be ported to/from other projects

Code style

- Every language has it’s own accepted style guide(s)
  - Consistent reading experience
  - Easy to recognize what code does
  - Don’t invent your own style

- Examples
  - Python – PEP8
  - R – Google Style or Advanced R

Code style – PEP8

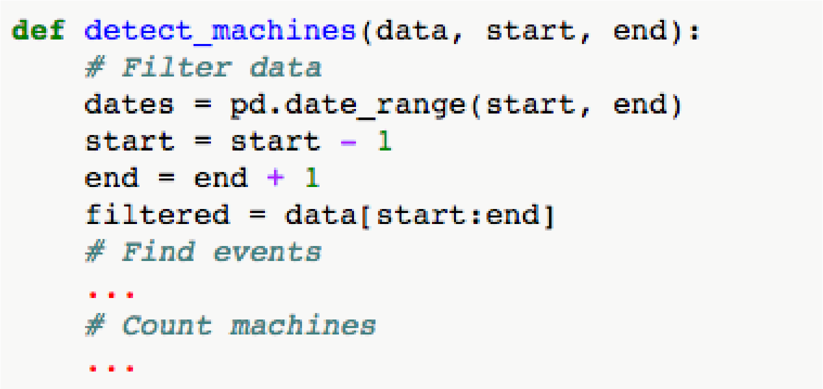
- Examples
  - Functions/variables `lower_case_variable`
  - Classes `UpperCaseClass`
  - Whitespace
    ```
    do_this(whitespace, next, to, commas,
            and, proper, indentation)
    ```

- Many useful tools
  - Style checkers – Flake8, Pylint
  - Automated formatters – YAPF, Black

Programming principles

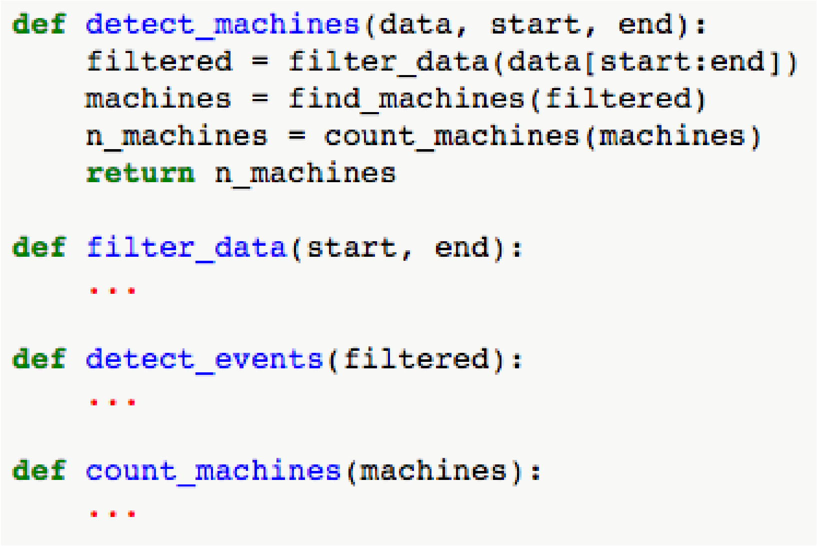
- Do one thing and do it well
  - Have small, focused functions/classes that only do one thing
  - Functions should be logical units

👎



```python
def detect_machines(data, start, end):
    # Filter data
    dates = pd.date_range(start, end)
    start = start - 1
    end = end + 1
    filtered = data[start:end]
    # Find events
    ...
    # Count machines
    ...
```

👍



```python
def detect_machines(data, start, end):
    filtered = filter_data(data[start:end])
    machines = find_machines(filtered)
    n_machines = count_machines(machines)
    return n_machines

def filter_data(start, end):
    ...

def detect_events(filtered):
    ...

def count_machines(machines):
    ...
```

Programming principles

Goal – write code other people (and future you) can understand

Don'ts

- Long functions doing multiple things
- Copy/paste code
- Re-use variable names in function
- One-char variables, abbrevations
- Variables that differ by one character
- Long, complicated variable names
- Many temp vars (or using your own defaults)
    - `bassie, buh, zip`

Do's

- Small functions doing one thing
    - `check_boiler()`
    - `load_rankings()`
- Build libraries, functions, classes
- Follow existing design patterns
- Descriptive and concise variables:
    - `male_user, is_fridge`
- Common temp vars
    - `temp, df`

Additional resources

Software development skills for data scientists:

- http://treycausey.com/software_dev_skills.html

Machine learning in production

- http://www.slideshare.net/turi-inc/machine-learning-in-production

Some Design Patterns for Real World Machine Learning Systems

- http://www.slideshare.net/justinbasilico/is-that-a-time-machine-some-design-patterns-for-real-world-machine-learning-systems

Documentation

GO
DATA
DRIVEN

Documentation

- One way to improve the readability of your code is to add (proper) documentation

- Different documentation types
  - Inline comments - explain what specific pieces of code do
  - Docstrings - document Python functions/classes/modules
  - Actual documentation – how to install, usage guide, etc.

Comments

- Avoid adding comments explaining the obvious
- Think about the choices/assumptions you make in your code, which are not directly clear from the code itself

👎

```
# import packages
import pandas as pd

# load some data
df = pd.read_csv('data.csv', skiprows=2)
```

👍

```
# Data contains two lines of description
# text, skip to avoid errors.
df = pd.read_csv('data.csv', skiprows=2)
```

Docstrings

- Docstrings document how to use specific functionality

```
def rescale_between(array1d, lower=0.0, upper=5.0):
    """Rescales array values between given upper/lower bounds.

    :param np.ndarray array1d: Values to be rescaled.
    :param float lower: Lower bound of the rescaled values.
    :param float upper: Upper bound of the rescaled values.

    :returns: Array containing rescaled values.
    :rtype: np.ndarray
    """
    …
```

- Can be accessed using `help(…)` (or ? in IPython/Jupyter)

Sphinx

- Sphinx is the de-facto tool to use in Python for writing and generating docs
- Docs are written in reStructuredText
- Many features
  - Hierarchical structure, table of contents, etc.
  - Generating docs from docstrings
  - Different themes, output types (html)

Sphinx – getting started

- Getting started
    - Install sphinx using `pip install sphinx`
    - Generate initial template using `sphinx-quickstart`
    - Start editing your docs
    - Build your docs using `make html` (in the docs folder)
- Some templates (cookiecutter-pypackage) include an initial structure that you can use

Testing

Testing

- Why test?
  - Confirm that your code does what you expect
  - Prevent regressions (code changes that change behavior)
- Two (main) types of tests
  - Unit tests – tests a single function/method
  - Integration tests – tests behavior of combined functions

Testing frameworks

- Python has multiple testing frameworks
  - Unittest – builtin framework, inspired by JUnit
  - Nose/Pytest – popular third party libraries
- We will focus on [Pytest]
  - Easy to use, with little boilerplate code
  - Can be a bit ‘magic’ in the beginning

GO
DATA
DRIVEN

Test structure

```
├── my_package
|  ├── __init__.py
|  ├── helpers.py
|  └── utils.py
└── tests
|    ├── conftest.py
|    └── my_package
|      └── test_helpers.py
├── setup.py
└── ...
```

Package code

Tests mirror
package structure

A simple example

```python
# test_helpers.py

from my_package.helpers import add_two

class TestAddTwo:
    """Tests for the add_two helper function."""

    def test_positive(self):
        """Tests addition with a positive number."""
        assert add_two(1) == 3

    def test_negative(self):
        """Tests addition with a negative number."""
        assert add_two(-3) == -1
```

Fixtures

- Fixtures allow you to define functions that setup elements required by (multiple) tests

```
import pytest
import smtplib

@pytest.fixture(scope="module")
def smtp_connection():
    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)

def test_smtp(smtp_connection):
    ...
```

Running pytest

```
$ pytest
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item

test_sample.py F [100%]

================================= FAILURES =================================
_________________________________ test_answer _________________________________

def test_answer():
> assert inc(3) == 5
E assert 4 == 5
E + where 4 = inc(3)

test_sample.py:6: AssertionError
========================= 1 failed in 0.12 seconds =========================
```

When to stop testing?

- So when do we have enough tests?
  - Ideally – when our code is bug-free
  - In practice – when we have ‘enough’ confidence in our code

- A popular metric is code coverage
  - Percentage of code covered by tests
  - Note: code with 100% coverage is not bug-free

- Can be generated in pytest using `pytest-cov` plugin

Building web API’s using Flask

GO DATA DRIVEN

Web theory – methods

- There are different types of request methods:
  - GET - Retrieve the resource from the server
  - POST - Create a resource on the server
  - PUT - Update the resource on the server
  - DELETE - Delete the resource from the server

- In general; you should keep GET requests limited to requests that do not change the state of the server.

Web theory – response types

- The status of a HTTP request is indicated using a code:
  - 1xx - continue
  - 2xx - you got a response
  - 4xx - server thinks a client made an error
  - 3xx - redirect
  - 5xx - server thinks that it made an error

- We're omitting a lot of details now and a full summary can be found here.

Flask

“Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions.”

- A minimal example:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!', 200
```

```
$ export FLASK_APP=hello.py
$ python -m flask run
```

Flask – class-based approach

```
from flask import Flask

class App(Flask):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.add_url_rule(
            "/",
            view_func=self.hello_world,
            methods=["GET"])

    def hello_world(self):
        return "Hello world!", 200
```

Flask is single-threaded

- Flask starts a Python process which is a single thread.
  This means it can only handle one request at a time.

- You must wrap the application in a WSGI (Web Server Gateway Interface) to serve multiple clients simultaneously.

- For more details, take a look at deploying Flask in Production.

Making requests

*“**Requests** is the only Non-GMO HTTP library for Python, safe for human consumption.”*

```
>>> r = requests.get('https://api.github.com/user’)
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.text
'{"type":"User"...'
>>> r.json()
{'private_gists': 419, 'total_private_repos': 77, ...}
```

Containerization using Docker

Why containers?

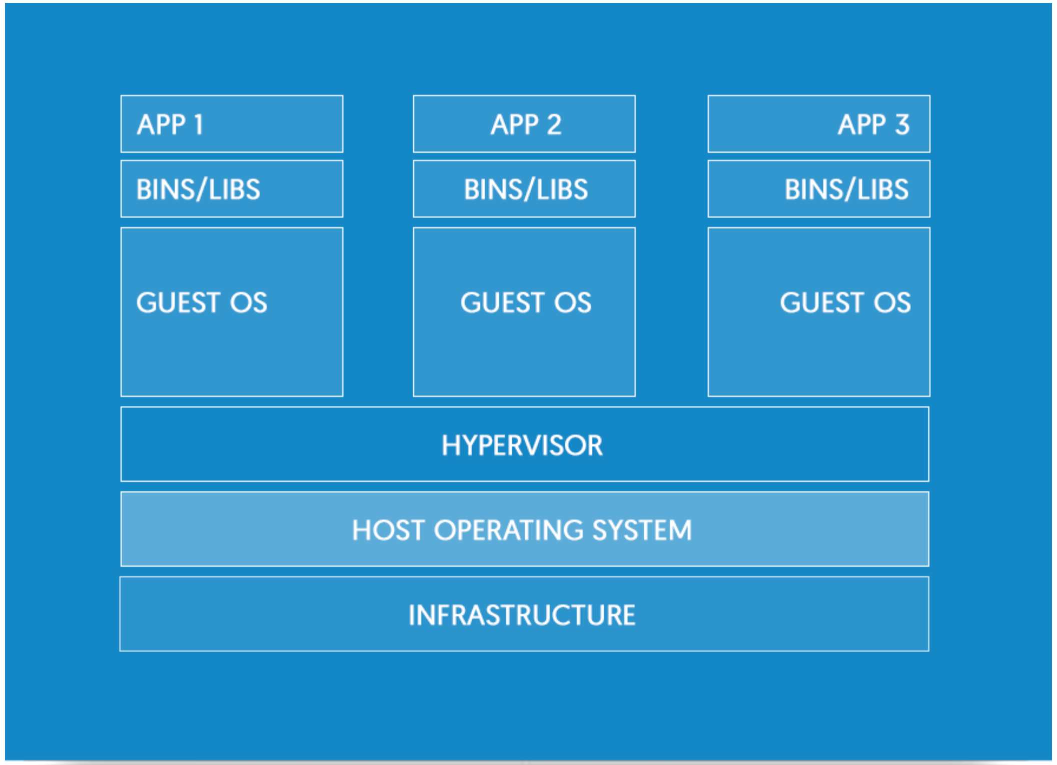
- Package your application
- Run it everywhere
- Lightweight
- No more dependency hell

Containers vs. VM’s

- VMs run on top of a hypervisor
  - Includes an OS
  - Dependencies
  - Your application

- Containers
  - Run on a layered filesystem
  - Run as an isolated process
  - Share the kernel

Containers vs. VM's



APP 1 | APP 2 | APP 3

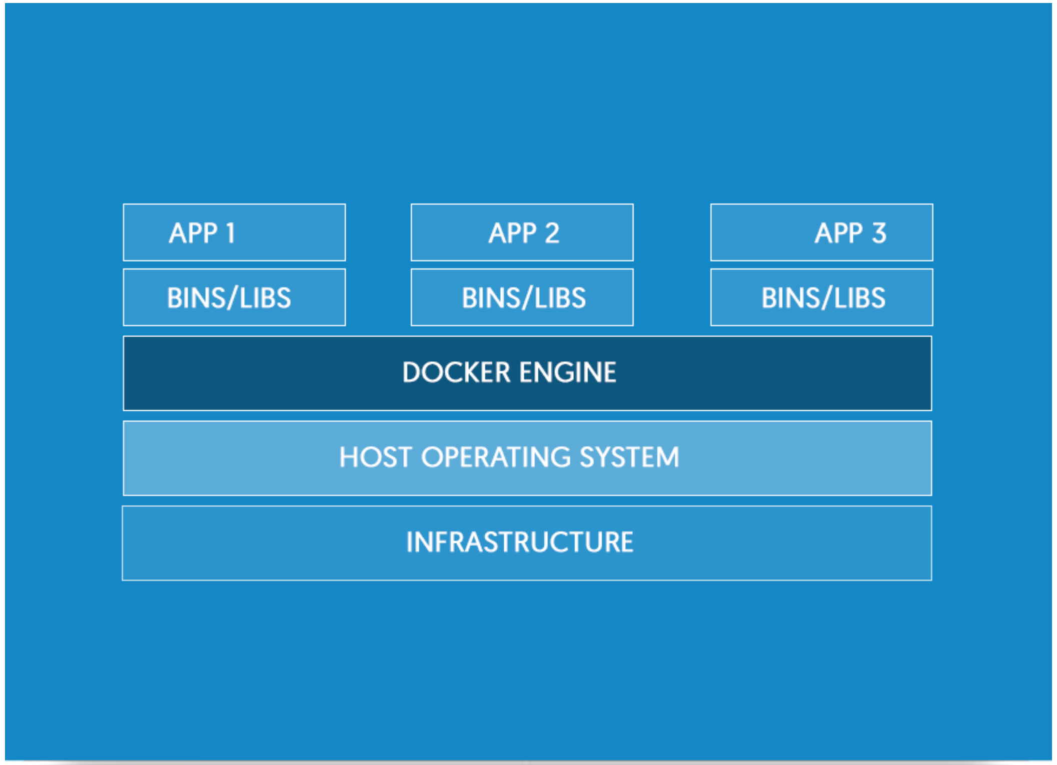
BINS/LIBS | BINS/LIBS | BINS/LIBS

GUEST OS | GUEST OS | GUEST OS

HYPERVISOR

HOST OPERATING SYSTEM

INFRASTRUCTURE



APP 1 | APP 2 | APP 3

BINS/LIBS | BINS/LIBS | BINS/LIBS

DOCKER ENGINE

HOST OPERATING SYSTEM

INFRASTRUCTURE

Docker

- Most popular container engine

- Provides an easy CLI to create and manage containers

- Rapid development
  - New version every ~3 months

Dockerfile example

```
FROM ubuntu:16.04
MAINTAINER Sven Dowideit <SvenDowideit@docker.com>

RUN apt-get update && apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo 'root:screencast' | chpasswd
RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/ssh/sshd_config

# SSH login fix. Otherwise user is kicked off after login
RUN sed 's@session\s*required\s*pam_loginuid.so@session optional pam_loginuid.so@g' -i /etc/pam.d/sshd

ENV NOTVISIBLE "in users profile"
RUN echo "export VISIBLE=now" >> /etc/profile

EXPOSE 22
CMD ["/usr/sbin/sshd", "-D"]
```

Questions?

GO
DATA
DRIVEN